Experience in Implementing a Parallel File System*

Rolf Riesen† Arthur B. Maccabe‡ Stephen R. Wheat§

March 1993

Abstract

With ever increasing processor and memory speeds, new methods to overcome the "I/O bottleneck" need to be found. This is especially true for massively parallel computers that need to store and retrieve large amounts of data fast and reliably, to fully utilize the available processing power.

We have designed and implemented a parallel file system, that distributes the work of transferring data to and from mass storage, across several I/O nodes and communication channels.

The prototype parallel file system makes use of the existing single threaded file system of the Sandia/University of New Mexico Operating System (SUNMOS). SUNMOS is a joint project between Sandia National Laboratory and the University of New Mexico to create a small and efficient OS for Massively Parallel (MP) Multiple Instruction, Multiple Data (MIMD) machines.

We chose file striping to interleave files across sixteen disks. By using source-routing of messages we were able to increase throughput beyond the maximum single channel bandwidth the default routing algorithm of the nCUBE 2 hypercube allows. We describe our implementation, the results of our experiments, and the influence this work has had on the design of the Performance-oriented, User-managed, Messaging Architecture (PUMA) operating system, the successor to SUNMOS.

*This research was supported in part by Sandia National Laboratories under contract ??.???.???

†Department of Computer Science; The University of New Mexico; Albuquerque, NM 87131; email: riesen@cs.unm.edu

‡Department of Computer Science; The University of New Mexico; Albuquerque, NM 87131 email: maccabe@cs.unm.edu. Currently on sabbatical at Sandia National Laboratories.

§Organization 1424; Sandia National Laboratories; Albuquerque, NM 87185-7761; email: srwheat@cs.sandia.gov

1 Introduction

Massively Parallel (MP) distributed memory systems are used to run large-scale scientific applications. While the number of processing elements easily scales for ever larger problem sizes, this is not generally true for secondary storage. If an application is given more processing power to solve a larger problem size, its I/O requirements may increase, using the same number of I/O devices. Vendors of Multiple Instruction, Multiple Data (MIMD) machines usually allow the connection of multiple disk subsystems to various parts of the machine. However, with a traditional file system, applications have to be modified to take advantage of the added capacity. An application tuned to a particular topology and configuration will be hard to port to another architecture. The goal of this project was to increase throughput to secondary storage transparently to the application.

The need to access mass-storage quickly stems from the fact that typical scientific applications produce large amounts of data. For example, a program running on 1024 processors for 60 minutes, could easily produce 1MB (mega byte) of data on each node. This program would need 1GB (giga byte) of disk space to store the final result. At the beginning of this project it took more than 43 minutes to transfer that much data to a file using the `fwrite` function. Applications that need to store intermediate results on secondary storage because there is insufficient memory available on each node, may experience additional performance decreases. For these applications the ratio of computation to I/O time can easily drop below one.

The goal of this project was to increase throughput when accessing data on secondary storage. We wanted to use multiple disks simultaneously to increase throughput beyond the data transfer rate of a single disk. Furthermore, we wanted to exploit the capability of the nCUBE 2 hardware to simultaneously send and receive data through more than one DMA channel. In Section 2 we show the hardware and software environment we had available to develop and test our system. Section 3 describes the communication primitives used to implement the file system. Section 4 discusses the basic, sequential file system upon which the parallel file system was built. In Section 5 we show the performance improvements attained by tuning parameters in the file system and the device driver. Then, in Section 6, we explain how we implemented the parallel file system. Section 7 discusses the experience with a first implementation that used a single channel to the disks. Section 8 shows the measurements of a second implementation that uses multiple channels. Section 9 explores the tunable parameters in our parallel file system. Section 10 finally draws some conclusions and explains the impact of this research on the design of our next operating system.

2 Development and Test Environment

2.1 Hardware

Figure 1 shows the system configuration used to develop and study the parallel file system. An nCUBE 2 hypercube with 1024 array (or compute) nodes provided the foundation for this research. Attached to the 1024 array nodes are sixteen I/O nodes. Each I/O node is connected to a SCSI-2 disk controller that can handle up to seven disks. In our configuration,

each controller had access to a single 1GB disk. The I/O nodes have the same physical characteristics as the array nodes. All nodes have 4MB of memory, a Complex Instruction Set Computer (CISC) microprocessor, and 13 bidirectional Direct Memory Access (DMA) channels. Each I/0 node is connected to eight array nodes in the hypercube.

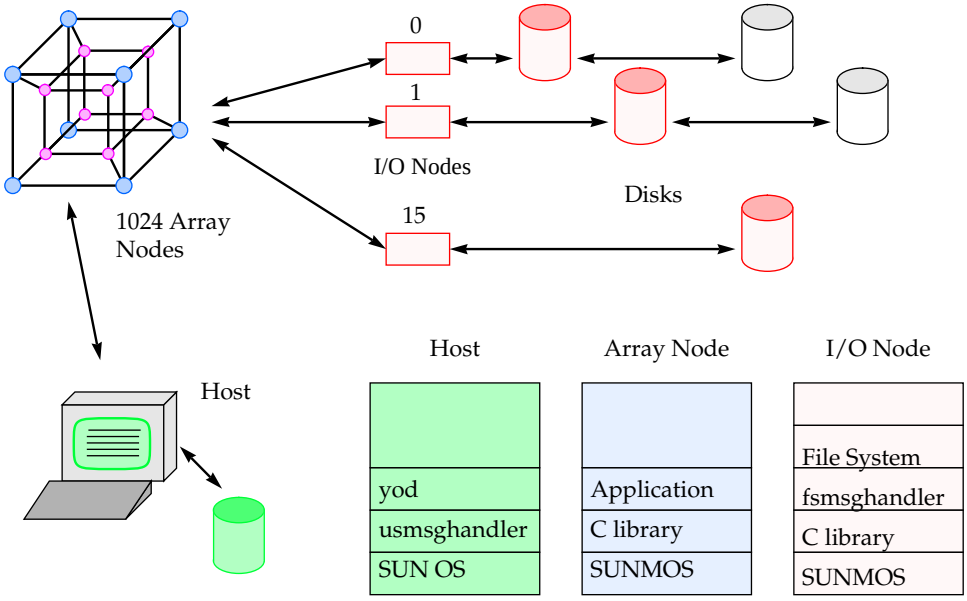

Figure 1: System Configuration

All connections within the cube, as well as the channels to the disks, have a bandwidth of 2.2MB/s. The DMA controllers are capable of driving all channels simultaneously. Furthermore, each connection is full duplex; that is, data can leave a node at 2.2MB/s while data is arriving from the same destination at the same rate [3].

If data must pass through intermediate nodes to reach its destination, the hardware uses wormhole routing. No software intervention or intermediate storage is needed. This greatly improves communication performance [1] [4].

A Sun workstation serves as host computer. It controls the operation of the cube; for example booting the operating system, and loading application and server programs.

2.2 Software

Tests and development were done under the Sandia/University of New Mexico Operating System (SUNMOS). SUNMOS is a small, efficient operating system for distributed MIMD architectures. In addition to other facilities, SUNMOS provides an emulation of Vertex, the vendor supplied operating system for the nCUBE 2. SUNMOS and the research described in this paper have greatly influenced the design of the Performance-oriented User-managed Messaging Architecture (PUMA) operating system, the successor to SUNMOS.

All array and I/O nodes run a separate instance of the SUNMOS kernel. Application programs running on the nodes are linked with the C language library that interfaces them

to the kernel. In addition to the standard C functions, the library provides routines that provide access to the inter-processor network. Two important functions are the `nuwrite` and `nuread` pair (see Section 3). These functions allow message passing between nodes and provide the foundation of all I/O operation.

The application program running on the I/O nodes is *fs*, the file system. The SUNMOS file system is a modified version of the MINIX file system [6] [7]. The file system accepts messages from array nodes to initiate I/O operations to the disk controllers. The SUNMOS file system offers services that look to the programmer (through the C library) like the standard Unix file I/O system. Specifically `fwrite` and `fread` allow I/O from array nodes to any disk attached to the cube.

The application loader program, *yod*, running on the host, facilitates stdio operations for applications running on the cube, as well as access to disk volumes available to the host.

The parallel file system described in this report is built on top of the SUNMOS file system. This facilitated the rapid prototyping of new ideas for experimentation. For PUMA we will rewrite the SUNMOS file system and make parallel access the default.

3 Communication Primitives

The lowest level functions to transfer data between nodes are `nwrite` and `nread`. These are Vertex compatible, blocking message passing functions. By blocking we mean that these functions do not return until the data has been transferred from or to the user provided buffer space.

SUNMOS maintains a buffer, called the *comm space*, to store incoming and outgoing messages. The size of the comm space can be set for each application but remains fixed during application execution. Comm space flooding can occur when more messages arrive than the comm space can hold. If flooding occurs, messages may be discarded. This is especially troublesome for servers that cannot anticipate the number or size of requests that will arrive in a given time frame.

In SUNMOS, the functions `nuwrite` and `nuread` behave similarly, but return immediately to the user. It is the user's responsibility to check a flag to determine when the transfer has been completed. Until that time, the buffer space should not be disturbed; otherwise, the data content might become corrupted. These non-blocking calls make it possible to continue processing while the DMA hardware performs the data transfer. Our parallel file system makes extensive use of this capability in order to parallelize transfers and manage the data stripes on the individual disks.

Two other important functions in SUNMOS are `readmem` and `writemem`. In the next section we will see that these functions allow us to build an efficient and reliable server protocol that is not subject to comm space flooding. Using `readmem`, a server can read data directly from the application's memory on another node. The `writemem` function allows a server to write data directly into an application's buffer, thereby circumventing comm space.

4 File System

The SUNMOS file system uses a message handler on each I/O node. The C library functions used by applications running on the array nodes are based on a simple Remote Procedure Call (RPC) protocol. These functions translate application requests (e.g., `fopen`, `fread`, and `fwrite`) into messages that are sent to the message handler on the appropriate I/O node. Depending on the message type, the message handler calls the corresponding function in the file system which then honors the request.

Earlier we mentioned that too many messages could flood the comm space of an I/O node. This is especially true for large file transfer requests that can easily exceed the size of the available comm space. Figure 2 shows the data flow for the original `fwrite` function.

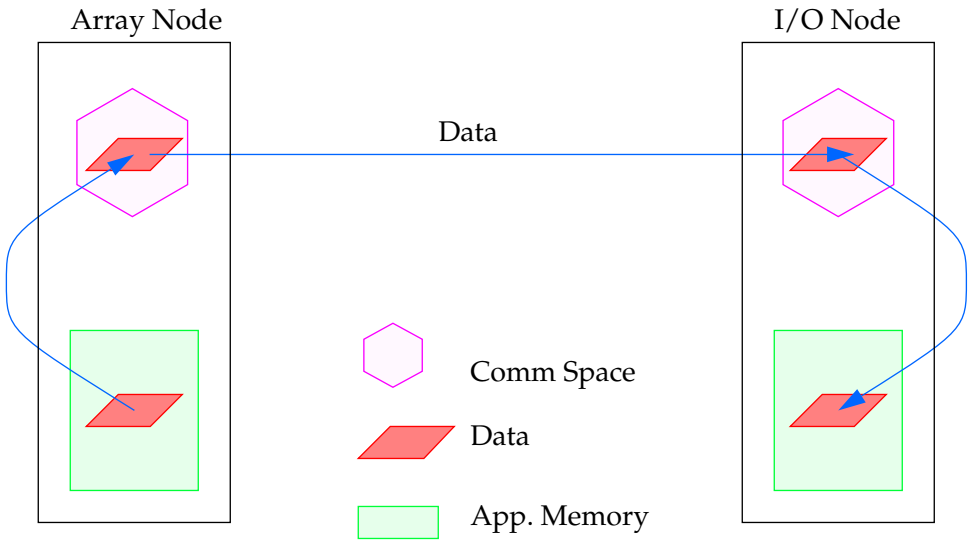

Figure 2: Data flow in the old `fwrite`

Besides the unreliability, this transfer method unnecessarily duplicates space on each node and hampers performance because of the required memory-to-memory copies. Before we could embark on building a parallel file system, we had to improve this protocol to ensure that data sent, arrived at its destination reliably and efficiently. Figure 3 illustrates the new protocol.

The C library, activated by a call to `fwrite`, sends a short request message to the I/O node using `nwrite`. This request contains the location of the data in user space, the amount of data to be transferred, and the file descriptor. The message handler uses this information to initiate a `readmem` request for a portion of the user data. This approach enables the message handler to use double buffering. While one packet is being sent to the disk, the file system can request another portion of the user data. Since the file system controls the data transfer, it can always make sure that there is space allocated for the requested data.

When the data has been read from the user space and sent to the disk, an acknowledgment message is sent to the library of the initiating application. Control is then returned to the application program.

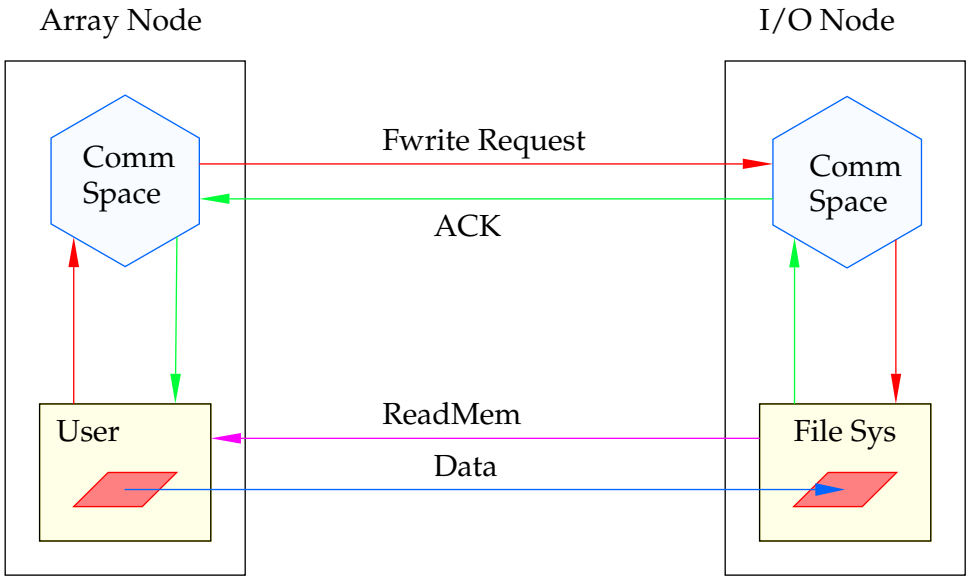

Figure 3: Data flow in the new `fwrite`

Note that only requests and acknowledgments are passed through the comm space. The data is transferred directly from user space into a file system buffer. Because the basic I/O operations (`fread` and `fwrite`) block the execution of the application program, each array node can have at most one outstanding data transfer request (SUNMOS is single tasking). This guarantees that the file system will never have more requests than the total number of nodes in the system. Since these requests have a fixed and small size, the file system can reserve enough comm space at startup.

A second advantage of the new protocol is scalability. On a larger machine there will be more nodes with, possibly, larger memories. Therefore, more requests for larger data sizes might arrive in the comm space of an I/O node. Since the maximum number of requests can be determined beforehand and is independent of the amount of data to be transferred, the new protocol scales easily. While the size of the comm space increases linearly with the number of nodes, the constant factor is very small.

The protocol for our `fread` is the mirror image of the `fwrite` protocol. Instead of the `readmem` function, the file system uses the `writemem` function to deposit the data directly into the application's memory.

5 Low-Level Tune-Up

Figure 4 diagrams the performance (throughput) of the `fwrite` function using the new protocol. For our tests we wrote files of increasing size to the disk. In particular, we wrote files, whose sizes were multiples of 8kB, from 8kB to 256kB. Additionally, we wrote files whose sizes were multiples of 128kB.

For one test, we transferred data into the cache on the I/O node and returned as soon as

the data was stored. For each file we cleared the cache. If a file fit completely into the disk cache, only a minimum number of disk accesses were necessary to read and update directory information.

The second test wrote the same files again (after they had been deleted), but this time the disk cache operated in write-through mode. Thus, the whole file was written to the disk before the operation completed. A third test measured the speed of `fwrite` under Vertex.

Using cache, the transfer rate approaches the hardware bandwidth of the node connections (2.2MB/s). When the cache becomes full, and data must be transferred to disk, throughput drops dramatically. The final throughput asymptotically approaches the 400kB/s we achieved when using the write-through option.

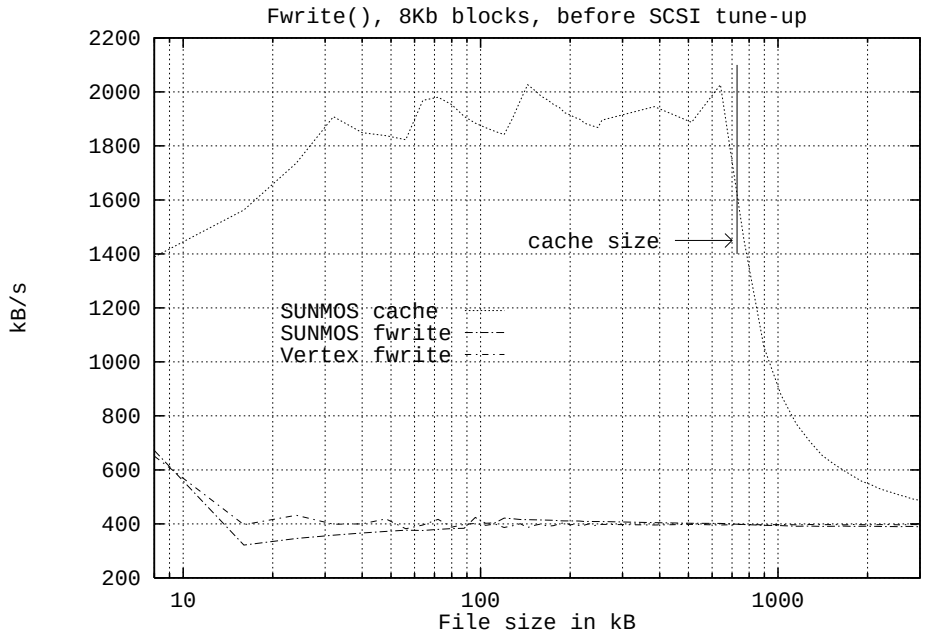

Figure 4: Before tune-up

Since we were going to build the parallel file system on top of the existing file system, we wanted it to perform as fast as possible. We identified the following parameters to tune performance:

Cache size. The unused portion of the memory on the I/O nodes is used for caching. With a block size of 8kB and 16kB we had 100 cache blocks for a cache size of 800kB and 1600kB respectively. When the block size was 32kB we had room for 80 blocks corresponding to a cache size of 2560kB.

SCSI transfer size. SCSI transfers data in disk-sector-size chunks. Our disks have a sector size of 512 bytes. The maximum number of sectors transferred with one command can be set between 1 and 127 sectors. Requests for larger transfers than the maximum setting are split into multiple transfers. Smaller transfer requests are performed in a single burst.

Block size. MINIX, like other UNIX file systems, uses blocks as the unit of disk transfer. Block sizes are typically between 512B and 32kB. Larger block sizes improve the transfer rate but waste disk space due to fragmentation. Disk fragmentation is a significant problem when many small files are stored on the disk. Smaller block sizes improve disk space utilization, but increase transfer time due to the repeated overhead [2].

Originally, the SCSI transfer size was set to 8 sectors (4kB) in our SCSI driver. As Figure 5 indicates, this resulted in a maximum transfer rate of about 400kB/s. After increasing the size to the maximum of 127, we were able to attain more than 1400kB/s for sufficiently large data transfers.

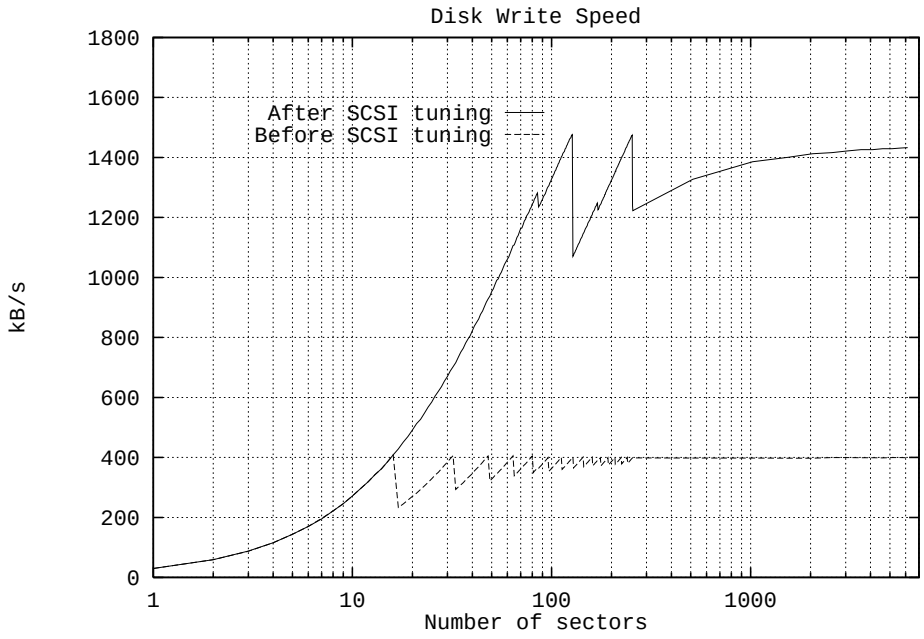

Figure 5: SCSI write throughput

Even after this improvement, the file system transfer rate remained the same. The reason for this is that the file system used a block size of 8kB (16 sectors) that was not large enough to exploit the improved transfer rate of the SCSI driver. Therefore, we increased the block size to 16kB and finally to 32kB. Figure 6 shows the result of these changes.

While the transfer rate into cache remained the same, the time it took to write a file to disk was significantly lowered by using 16kB blocks and even more so by using 32kB blocks. In a production environment, the cache will quickly become full. Modified data blocks will have to be written to the disk in order to make room for new blocks. Read requests might benefit from the availability of data in the cache; however, most writes will cause blocks to be written to disk. Therefore, the most interesting curve in Figure 6 is the middle one that shows throughput using the write-through method.

We chose a block size of 32kB because we wanted to tune our file system for large files.

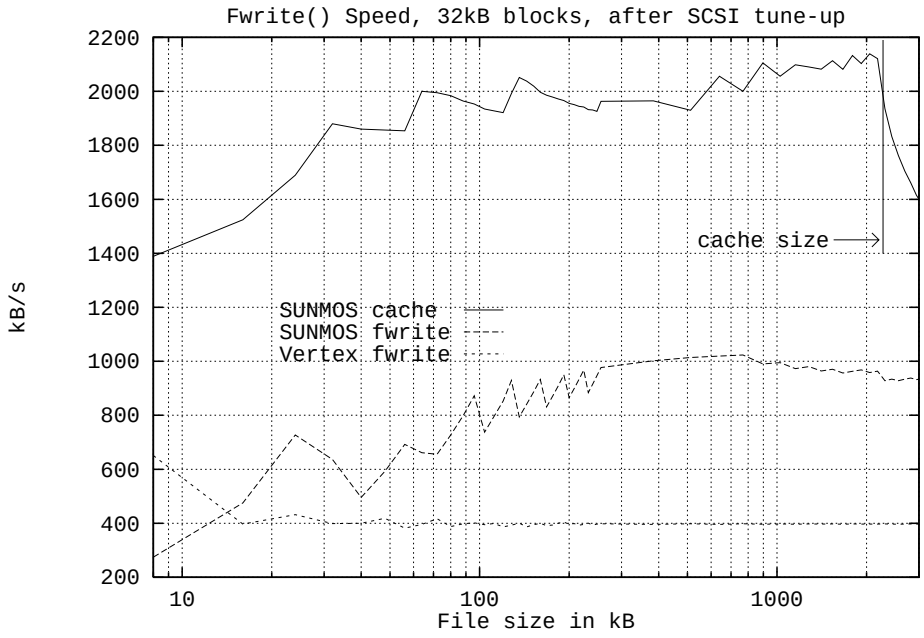

Figure 6: After SCSI tune-up with 32kB blocks

Selecting an even larger block size did not result in much higher throughput at the cost of significantly increased disk fragmentation.

6 Parallel Striping

Several methods have been proposed to increase disk throughput. Interleaving data across several disks sends the contents of a single file to multiple controllers and disks. The goal is to reduce data transfer time by a factor of $1/n$, where n is the number of disks. This assumes that there are n independent channels available, or the capacity of the channels is not a limiting factor [5].

Interleaving can be done at the bit, byte, sector, block, or record level. When interleaving is performed at the block level, it is often called striping. One stripe of data with a length corresponding to the block size is placed on the first disk. The second stripe goes to the second disk and so on until all disks have been used. If there are more stripes to store, the first disk is reused.

We chose striping at the block level for several reasons:

- It can be easily implemented on top of the existing file system as an additional library layer.
- As a prototype, the system should be easy to modify, so that new algorithms and parameters can be tested quickly without rebuilding the file system.

- If the chunks sent to individual I/O nodes are contiguous data regions of the original file, bookkeeping and management of individual stripes can be done with less overhead.

For this experimental version of our parallel file system, we collect all the stripes for one file on a given disk into a single file. An additional file is used to store information about the structure of the parallel file; e.g. stripe size, disks used, total size, size of each disk, etc. This strategy allows for consistency checks and increases the speed of opening a parallel file. Therefore, each parallel file is represented by $n+1$ regular files, where n is the number of disks spanned by the parallel file.

6.1 Example of a Parallel Write

In this section, we discuss a write operation to a parallel file. This example illustrates how the data is distributed among the disks. In this example, we assume the user has just issued a request to write a 64kB data block to a parallel file. We further assume that the file already exists and is distributed over three disks.

Our example uses a stripe size of 8kB. In Figure 7, we see that the file already contains 34kB of data—two full stripes on Disk 0, one complete stripe on Disks 1 and 2, and an incomplete stripe on Disk 1.

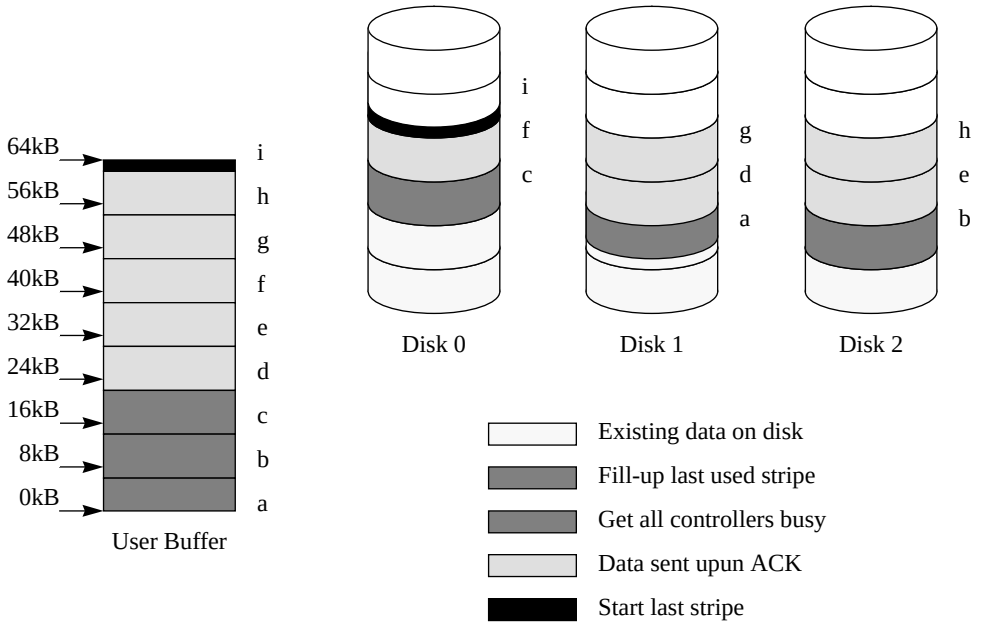

Figure 7: Add User Buffer to Existing Parallel File

We cannot simply divide the user buffer into 8kB blocks and send them to the disks. After a series of writes, it would be impossible to reassemble the original file. Instead, we have to proceed in the following manner:

- Find out how much more data fits into the last used stripe. Send that amount of data to the appropriate disk. In our example we send 6kB (block a) to Disk 1.

- To keep as many channels and disks busy as possible, we carve the remaining buffer into 8kB blocks and send one block to each disk that is not already receiving data. Disk 0 and Disk 2 in our example are still idle. So, we send them each an 8kB block (blocks b and c).
- We now wait for an acknowledgment signal from one of the I/O nodes. Whenever we receive one, and there is data left for that particular disk, we send another 8kB block. In our example we fill blocks d, e, f, g, and h.
- When blocks c and f have been written and we receive another acknowledgment signal from Disk 0, we send the partial block i.

After the first three blocks (a, b, and c) have been sent, operation proceeds asynchronously. Whenever an I/O node completes its task, we send it the next block of data. Thus, blocks d, e, f, g, and h are written in no particular order, with the exception, of course, that block d precedes block g, and block e precedes block h. This scheme keeps all I/O nodes and disks busy at the same time.

7 First Implementation

Figure 8 shows throughput measurements for the first implementation of the new parallel `fwrite`. Compared to the underlying sequential `fwrite`, there is practically no improvement. While this is not what we had expected, it does indicate that the overhead of managing the individual stripes is not significant.

The reason for the poor performance is the default routing algorithm that determines the path taken by messages sent from an array node to an I/O node. The default algorithm used in the nCUBE 2 computer is the E-cube routing algorithm. It is a dimension-ordered routing method that uses minimal path length and is guaranteed to be deadlock free [4]. If messages are sent to different locations, they might share a common channel, depending on the position of the originating node relative to the destination. Figure 9 illustrates the use of a common channel. In this case, the fanout (use of multiple channels) does not occur until the first hop has been completed through a common channel.

8 Second Implementation

The nCUBE 2 hardware supports source routing where the originating node places information about the entire path into the header of the message. This allows us to force packets out more than one channel and reduce channel contention. We do this in a round-robin fashion using the four upper channels on the array nodes. We specify a direct neighbor as the first destination for each packet. From that neighboring node, we let the packet travel along the default path. This guarantees that our new scheme is deadlock free, while reducing contention for common channels.

Note that our strategy may result in packets using longer routes than in the default routing algorithm. The default routing algorithm brings a packet closer to its destination after

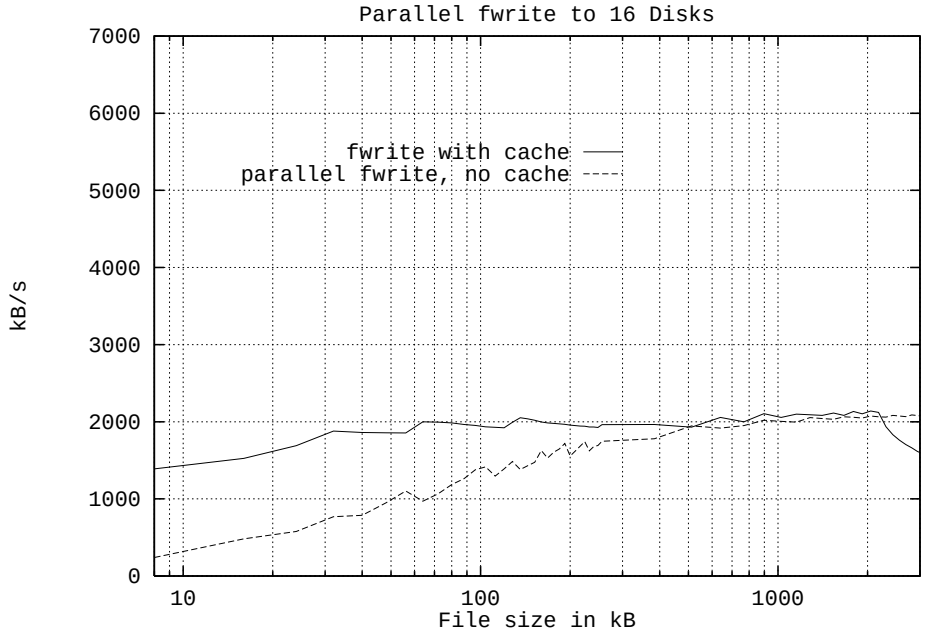

Figure 8: Throughput vs. File Size

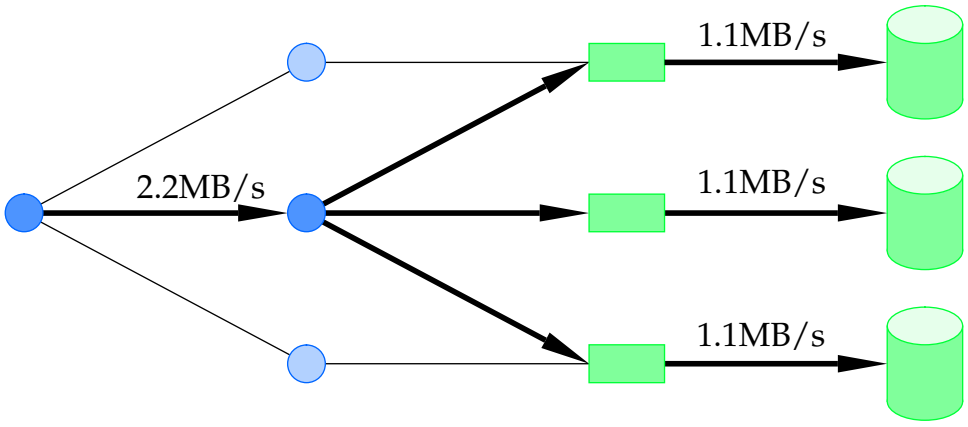

Figure 9: Single channel due to late fanout

each channel it traverses. Our strategy introduces at most two extra hops into the path of a packet, but we gain a fourfold increase in the throughput of data leaving a node (see Figure 10). These additional hops increase latency, but this increase is more than compensated for by the higher throughput.

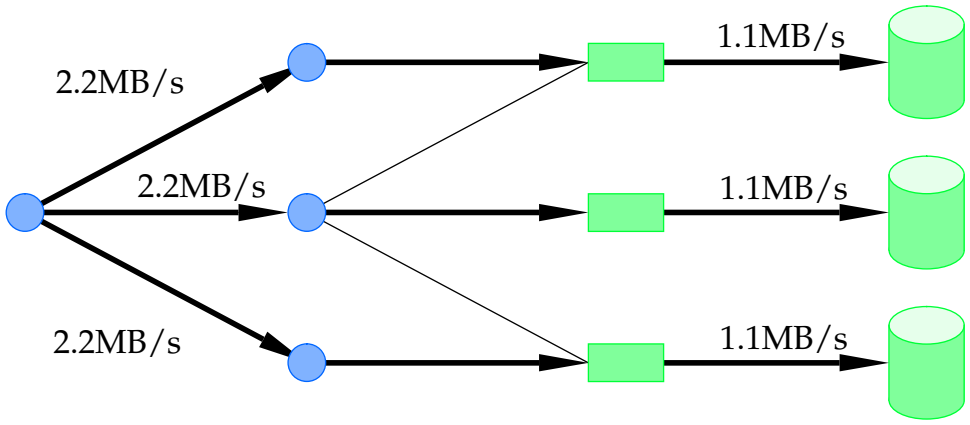

Figure 10: Multiple channel due to early fanout

Our experience indicates that it does not pay to use more than four channels simultaneously. Mainly because the DMA/memory hardware is limited to 10-12MB/s. In addition, channel contention increases when more channels are used. Using one to four channels we observed a steep increase in throughput. While the throughput continued to increase when using more channels, it began to level off after four channels.

Figure 11 shows the performance of the new parallel `fwrite` using multiple channels and compares it to the single channel performance. It is interesting to note the irregularity of the curve when multiple channels are used. Let us consider the valley at the 80kB mark to show what circumstances led to this particular drop in performance.

To send 80kB of data, the first stripe (16kB) is sent out the first channel to the appropriate disk. Then the second stripe is sent out the second channel and so on. After the transfer of four stripes (64kB) has been initiated, all four channels are busy. However, the parallel file system continues to generate requests. It tries to send out the last stripe and has to reuse the first channel to do so. Since the transfer of the very first stripe is still in progress, transmission of the last stripe is delayed.

Figure 12 illustrates that sending five stripes takes twice as much time as sending four stripes, due to channel contention. In other words, we could have sent eight stripes in the same time it took to send five stripes. The irregularities in Figure 11 are due to varying utilization of the available channels.

9 Parallel File System Parameters

Using our parallel file system we were able to identify three tunable system parameters.

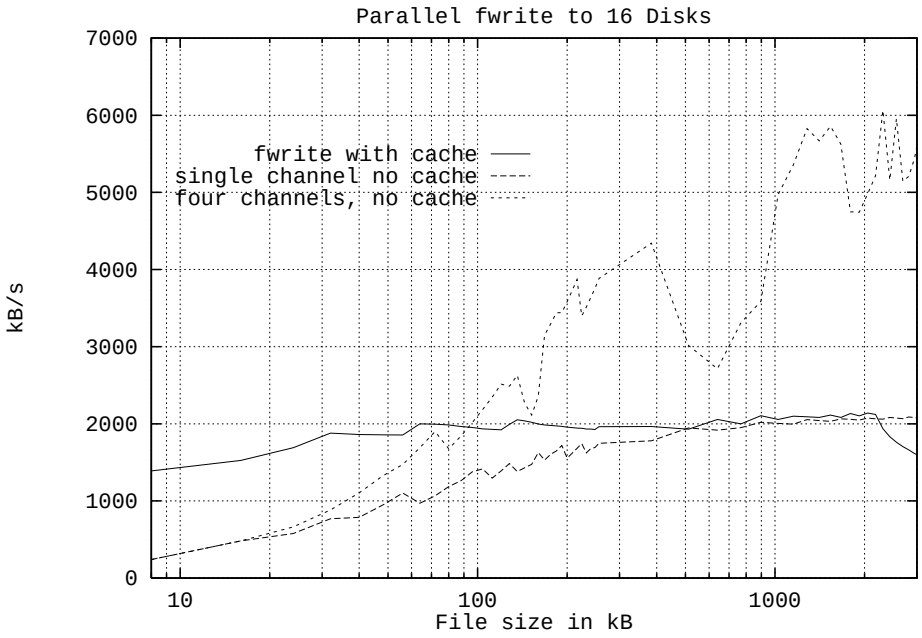

Figure 11: Throughput of parallel `fwrite`

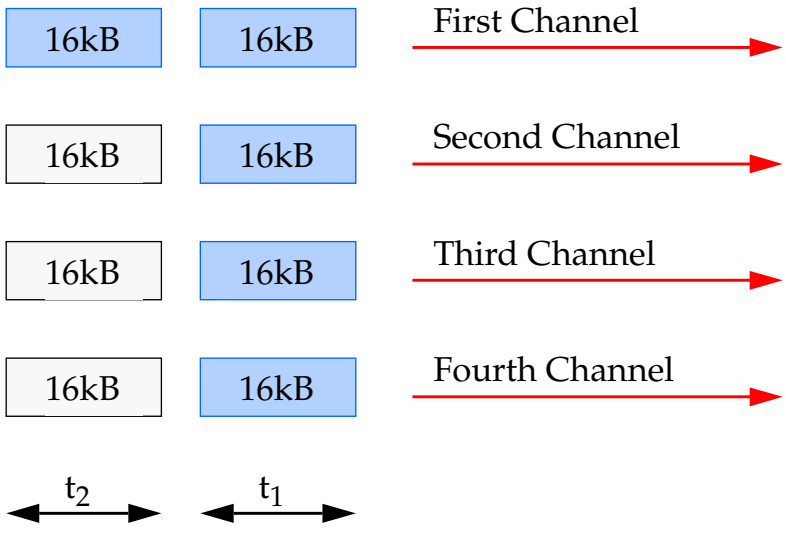

Figure 12: Cause of channel contention

Number of paths. By using more and more paths we are able to linearly increase throughput with every additional path used, up to four paths. Using more than four paths further increases throughput, but the increase is no longer linear. The DMA memory bandwidth of the processing elements limits the total maximum throughput to between 10MB/s and 12MB/s.

Using more than four channels has the disadvantage that we interfere with message transmissions of neighboring nodes without a significant gain. Also, by "reserving" the full memory bandwidth for I/O, processing on that node is impeded.

Number of disks. Since we are using only four channels, striping a file across more than four disks does not pay off. An application can partition the sixteen available disks into groups of four. It regards each group as a single device by using our parallel file system. If the application uses 64 nodes or more, it is possible to keep the data streams to each group completely separated.

Stripe size. If the size of each data transfer to a file for an application could be predetermined, then the optimal stripe size would be the size of the most frequently transmitted size divided by the number of available channels. If c is the number of channels that can be used, then a transfer request of size s would be split into c stripes of size s/c. The overhead to schedule and transfer the stripes would be at a minimum, while the throughput would be maximized.

The stripe size should not be too big, since that reduces channel utilization. If a small stripe size is chosen, the overhead of processing might interfere with throughput. We have observed that a stripe size between 8kB and 24kB is best when four channels are used.

10 Conclusions and Further Work

We have implemented our strategies as a set of C library function calls that closely mimic the standard UNIX file I/O interface. The management of channels, disks, and the individual stripes are transparent to the user. A parallel file in our system looks like a sequential file to a user.

From the start of this project to the current state, we have been able to increase throughput twenty-fold using a variety of techniques:

1. Tuning the parameters of our device driver to take advantage of the SCSI-2 interface.
2. Tuning the block size of our file system to match the optimal transfer size of the SCSI-2 interface.
3. Striping files across multiple disks to increase throughput.
4. Using the multiple channels available on the hypercube to further increase throughput.

Item 1 is a simple matter of analyzing performance and tuning the parameters to the given hardware configuration. Increasing the file system block size is a compromise. It benefits

the transfer of larger files, while wasting disk space due to fragmentation for smaller files. The applications run on these types of computers warrant, in our opinion, the fine-tuning of system parameters to benefit large file transfers.

File striping is a method that finds more and more acceptance in high performance computing. Not all applications will benefit form file striping. Some scientific codes do calculations and disk transfers in a lock-step fashion; either all nodes compute or do transfers. In that case the advantage of using multiple channels will be nullified since the node to disk transfers are interfering with each other. However, these type of applications will not see a performance degradation since the administrative overhead in our system is minimal. Some of these applications might even benefit since our striping approach packetizes the data and can help to better utilize the available channels. More research is required in this area.

The basic protocol underlying all our disk transfers is reliable and efficient. The use of the `readmem` and `writemem` functions poses a security risk, since any process on any node can read and modify any other application's memory. We are addressing this issue while implementing our system under PUMA.

When porting PUMA to a mesh architecture, the issue of using multiple channels will also have to be addressed. Performance measurements will tell us if it is worthwhile to consider multiple routes in such an environment.

11 Acknowledgments

We wish to thank Lisa Kennicott who helped us design the basic data transfer protocol underlying our `fwrite` and `fread` functions. The SUNMOS team was helpful with comments and ideas during the design phase and the preparation of this paper. We also wish to thank Sandia National Laboratories for the use of their equipment.

References

[1] Sergio Felperin, Prabhakar Raghavan, and Eli Upfal. A theory of wormhole routing in parallel computers. In *33rd Annual Symposium on Foundations of Computer Science*, pages 563–572, New York, N.Y., 1992. IEEE.

[2] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[3] nCUBE, 1825 NW 167th Place, Beaverton, OR 97006. *nCUBE 2 Processor Manual*, December 1990. PN 101636.

[4] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2), Feb 1993.

[5] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[6] Andrew S. Tanenbaum. *Operating systems : design and implementation.* Prentice-Hall software series. Prentice-Hall, Englewood Cliffs, N.J., 1987.

[7] Andrew S. Tanenbaum. *MINIX for the IBM PC, XT, and AT.* Prentice-Hall software series. Prentice-Hall, Englewood Cliffs, N.J., 1988.

A Manual Pages

NAME

pfopen – open a parallel file

SYNTAX

#include "clparfs.h"

PFILE *pfopen(char **filename*, char **mode*, int *last*, int **blk_size*);

DESCRIPTION

`pfopen` opens the parallel file *filename* and returns a pointer to a *PFILE* structure. This pointer is used to read, write, and close a parallel file. The method used to implement parallel files under *Sunmos* is file striping. A parallel file consists of a descriptor file and one to *MAX_STRIPES* stripe files. Each stripe file is located on a separate disk.

filename must be of the form:

```
//fdxy/pfname
```

x specifies the first controller to be used for the file, and y determines the disk on each controller. The descriptor file will be located on the disk attached to the first controller.

mode is one of the following:

r	open parallel file for reading (file must exist).
w	truncate or create a parallel file for writing.
a	open parallel file for writing at end of file, or create for writing.
r+	open for reading and writing.
w+	truncate or create for reading and writing.
a+	open or create for reading and writing at end of file (append).

last specifies the last controller to be used for this file. Note, that this parameter together with x in the *filename*, determines how many disks the parallel file will spawn. The value for *last* must be between x and *MAX_STRIPES*:

$$0 <= x <= last < MAX\_STRIPES$$

If the file already exists, the value of *last* is ignored. If a new file is created and the value of *last* is outside of the allowed range, then the default of *MAX_STRIPES* - 1, is used.

The parameter *blk_size* specifies the block size (thickness of each stripe) in bytes. The default *BLOCK_SIZE* (defined in fsconst.h) is used, if *blk_size* is less than one. This value is ignored, if the file already exists. The value chosen for *blk_size* should be the size of an average *pfread* or *pfwrite* divided by the number of disks the parallel file spawns.

RETURN VALUE

`pfopen` returns NULL if the operation fails. *pfs_error* contains the appropriate error number.

EXAMPLES

```
PFILE *pfp;
pfp= pfopen("//fd80/parfile", "w", 15, 0);
```

opens or creates the parallel file *parfile*. The file will be located on disk 0 of controllers 8 to 15; i.e. it will spawn eight disks.

```
PFILE *pfp;
pfp= pfopen("//fd02/parfile", "r", -1, 0);
```

opens the file *parfile* for reading. All parts of this file will be on disk 2 of each controller. The file must already exist. Therefore, the block size and the last controller in use are predetermined.

```
PFILE *pfp1, *pfp2;
pfp1= pfopen("//fd00/parfile", "w", 7, 0);
pfp2= pfopen("//fd80/parfile", "w", 15, 0);
```

These two files can coexist, while

```
PFILE *pfp1, *pfp2;
pfp1= pfopen("//fd00/parfile", "w", 11, 0);
pfp2= pfopen("//fd80/parfile", "w", 15, 0);
```

is an illegal combination, because the stripe files on controllers 8 to 11 are overlap-

ping.

SEE ALSO

pfclose(3), pfflush(3), pfwrite(3), pfread(3), pfremove(3), pfrename(3), pfeof(3), pferror(3), pfclrerr(3), pfperror(3), pfstrerror(3)

CAVEATS

It is very important to call `pfclose` before the program ends. If this is not done, the descriptor file will not be updated. Without the correct information in the descriptor file, the parallel file cannot be opened again, because the file positioning information has been lost.

AUTHOR

Rolf Riesen

NAME

pfclose, pfflush – close or flush a parallel file

SYNTAX

#include "clparfs.h"

int pfclose(PFILE **pfile*);
int pfflush(PFILE **pfile*);

DESCRIPTION

`pfclose` flushes any unwritten data to the parallel file *pfile*; i.e. the stripe files and the descriptor file. All automatically allocated buffers are freed and all open files are closed

`pfflush` flushes any unwritten data to the parallel file *pfile*; i.e. the stripe files and the descriptor file.

RETURN VALUE

`pfclose` and `pfflush` return zero on success and EOF on error.

CAVEATS

Unlike an ordinary file, a parallel file needs to be closed, or at least flushed, before the program terminates. Since this library is implemented on top of the existing Sunmos file system, a program terminating without a `pfclose` or `pfflush` , will leave the descriptor file un-updated.

When the parallel file is opened again, internal checks will detect a discrepancy between the actual file sizes of the stripe files and the numbers recorded in the descriptor file.

SEE ALSO

pfopen(3), pfwrite(3), pfread(3), pfremove(3), pfrename(3), pfeof(3), pferror(3), pfclrerr(3), pfperror(3), pfstrerror(3)

AUTHOR

Rolf Riesen

NAME

pfread, pfwrite – parallel file input/output

SYNTAX

#include "clparfs.h"

int pfwrite(void **ptr*, int *size*, int *nobj*, PFILE **pfile*);
int pfread(void **ptr*, int *size*, int *nobj*, PFILE **pfile*);

DESCRIPTION

`pfwrite` writes *nobj* objects of size *size* to the parallel file *pfile*. The data is taken from memory location *ptr*. `pfwrite` returns the number of objects actually written.

`pfread` reads at most *nobj* objects of size *size* from the parallel file *pfile*. The data is placed at *ptr*. `pfread` returns the number of objects actually read.

RETURN VALUE

`pfwrite` and `pfread` return the number of objects actually written or read. The return value might be different from *nobj*, if an error occurs; e.g. disk full. For some type of errors, the return value is 0; e.g. attempt to write to a write protected file.

The return value should always be compared to *nobj*. In case it is 0, the external variable *pfs_error* contains an appropriate error number. `pfperror` can be used to display the corresponding error text.

CAVEATS

While *ptr* is not required to point to any particular address, performance suffers, if *ptr* does not point to a word aligned object.

SEE ALSO

pfopen(3), pfclose(3), pfflush(3), pfremove(3), pfrename(3), pfeof(3), pferror(3), pfclrerr(3), pfperror(3), pfstrerror(3)

AUTHOR

Rolf Riesen

NAME

pfremove – remove (unlink) a parallel file

SYNTAX

#include "clparfs.h"

int pfremove(char **filename*, int *last*);

DESCRIPTION

`pfremove` removes all the stripe files and the descriptor file associated with the parallel file *filename*. If the descriptor file is readable, and appears uncorrupted, the parameter *last* is ignored. The information in the descriptor file is used to locate all stripe files and delete them.

However, if the descriptor file is corrupted, or does not exist, *last* is used to find the last stripe file. In this case, *last* should be set to the same value as the corresponding parameter in the `pfopen` command, when the file was created.

This feature allows the removal of partially corrupted parallel files; e.g. a parallel file with a missing descriptor file.

RETURN VALUE

`pfremove` returns zero on success and non-zero on error.

SEE ALSO

pfopen(3), pfclose(3), pfflush(3), pfwrite(3), pfread(3), pfrename(3), pfeof(3), pferror(3), pfclrerr(3), pfperror(3), pfstrerror(3)

AUTHOR

Rolf Riesen

NAME

pfrename – renames a parallel file

SYNTAX

#include "clparfs.h"

int pfrename(char **oldname*, char **newname*);

DESCRIPTION

`pfrename` renames or moves *oldname* to *newname*. The path of *newname* must already exist on all disks the parallel file spawns (see examples).

RETURN VALUE

`pfrename` returns zero on success and non-zero on error.

EXAMPLES

The following command renames the parallel file *//fd00/example* to *//fd00/new*:

```
pfrename("//fd00/example", "//fd00/new");
```

Let us assume for the moment, that *//fd00/example* had been created spawning controllers 0 and 1. The parallel file would then consist of the two stripe files *//fd00/.pfs.example* on disk 0 of controller 0 and *//fd10/.pfs.example* on disk 0 of controller 1. There would also be the descriptor file *//fd00/example* on disk 0 of controller 0.

The above `pfrename` command renames the two stripe files as well as the descriptor file. If renaming of any of these three files fails, `pfrename` will fail and return a non-zero value.

For the command

```
pfrename("//fd00/example", "//fd00/my_dir/new");
```

to work, the directory *my_dir* must exist on all disks which contain one of the stripe files.

SEE ALSO

pfopen(3), pfclose(3), pfflush(3), pfwrite(3), pfread(3), pfremove(3), pfeof(3), pferror(3), pfclrerr(3), pfperror(3), pfstrerror(3)

CAVEATS

`pfrename` is not atomic. If some of the stripe files, or the descriptor file, have been renamed, and the operation fails, `pfrename` tries to undo the changes. However, if another process has created a file that conflicts with *oldname* in the meantime, then the undo operation will fail.

RESTRICTIONS

oldname and *newname* must have the same controller and disk number; i.e. moving a parallel file from one disk to another is not allowed.

AUTHOR

Rolf Riesen

NAME

pferror, pfclrerr, pfeof – parallel file status inquiries

SYNTAX

#include "clparfs.h"

int pferror(PFILE **pfile*);
void pfclrerr(PFILE **pfile*);
int pfeof(PFILE **pfile*);

DESCRIPTION

`pferror` returns non-zero if the error flag for *pfile* is set.

`pfclrerr` clears the error and the end-of-file flag of *pfile*. Unless you close *pfile*, this is the only way to clear the error flag.

`pfeof` returns TRUE (non-zero) if the end-of-file flag is set.

NOTE

These functions are implemented as macros.

SEE ALSO

pfopen(3), pfclose(3), pfflush(3), pfwrite(3), pfread(3), pfremove(3), pfrename(3), pfperror(3), pfstrerror(3)

AUTHOR

Rolf Riesen

NAME

pfperror, pfstrerror – Parallel file system messages

SYNTAX

#include "clparfs.h"

void pfperror(char **str*);
char *pfstrerror(int *n*);

DESCRIPTION

`pfperror` prints the string *str*, followed by a collon, followed by a string describing the error number stored in *pfs_error*. If *str* is NULL, only the error message is printed. If *pfs_error* is zero (NOERR), then nothing at all is printed.

`pfstrerror` returns a pointer to a string describing the parallel file system error *n*.

pfs_error contains an error number describing the most recent problem encountered by the parallel file system. The currently implmented numbers and messages are:

ENOERR
: Illegal error number

EWRNGFMT1
: File name must begin with //fd

EWRNGFMT2
: File name must be of the form //fdxy/name

ENOMEM
: Out of memory

EWRNGCTRL
: Controller # must be: 0 <= CTRL < MAX_STRIPES

ENODESC
: Descriptor file does not exist

ESFEXISTS
: One of the stripe files already exists

ECREATDESC
: Can't create the descriptor file

EWRNGMAGIC
: Magic number read is incorrect

ECRPTDESC
: Corrupted descriptor file

EOPENSF
: Could not open stripe file(s)

EINVPAR1
: Parameter must not be NULL

ECLOSESF
: Error closing stripe file

EWRTDESC
: Error writing to descriptor file

ECLOSEDESC
: Error closing descriptor file

ENOUPDATE
: Can't open descriptor file for update

ENOWRT
: Can't open descriptor file for writing

ENOSF Stripe file does not exist

EINVPAR2
: size and nobj must be > 0

ERDONLY
: Parallel file is read only

EINVMODE
: Invalid mode

ERMDESC
: Can't remove the descriptor file

ERMSF Can't remove a stripe file

ENOREAD
: Can't open descriptor file for reading

EXAMPLES

The following example shows a possible application of `pfperror`.

```
if ((pfp= pfopen(fname, "w", last, bsize)) == NULL)   {
    pfperror(argv[0]);
    exit(-1);
```

```
}
```

SEE ALSO

pfopen(3), pfclose(3), pfflush(3), pfwrite(3), pfread(3), pfremove(3), pfrename(3), pfeof(3), pferror(3), pfclrerr(3)

AUTHOR

Rolf Riesen

B An Example Program

```
/*
** Parallel File System Example                          7/10/92 rr
**
** A simple program to show the use of some of the parallel file
** system function calls. Put this program into ~/ncube/sunmos/tests
** and compile it with:
**                  make partest.exe
**
** You can now run it:
**                  yod partest
*/



#include <clstdio.h>
#include "clparfs.h"



#define FILE_SIZE 100000 /* Max file size */
#define MAXRAND 32767.0 /* 2^15 - 1 */



void srand(int);
char send_buf[FILE_SIZE];
char recv_buf[FILE_SIZE];



/*
** main()
*/
int main(int argc, char *argv[])
{
    PFILE *pfp;
    int rcw, rcr, i;
    int cmp_err;

    /* The x in //fdxy/name determines the first disk to be used */
    char *fname= "//fde0/partest";

    /* Block (stripe) size */
    int bsize= 32 * 1024;
```

```
/* Last disk to be used */
int last= 15;



/* create random data in send buffer */
srand(ntime());
for (i= 0; i < FILE_SIZE; i++)
    send_buf[i]= rand() * 256 / MAXRAND;



/*
** Create and write a parallel file
*/
if ((pfp= pfopen(fname, "w", last, bsize)) == NULL) {
    pfperror(argv[0]);
    exit(-1);
}
rcw= pfwrite(send_buf, 1, FILE_SIZE, pfp);
if (pfclose(pfp) ≠ 0)
    pfperror(argv[0]);



/*
** Read the data back and delete the parallel file
*/
if ((pfp= pfopen(fname, "r", -1, -1)) == NULL) {
    pfperror(argv[0]);
    exit(-2);
}
rcr= pfread(recv_buf, 1, FILE_SIZE, pfp);
if (pfclose(pfp) ≠ 0)
    pfperror(argv[0]);
if (pfremove(fname, -1) ≠ 0)
    pfperror(argv[0]);



/*
** Compare input and output
*/
cmp_err= FALSE;
i= 0;
```

```
    while ((i < FILE_SIZE) && (!cmp_err)) {
      if (send_buf[i] ≠ recv_buf[i])
        cmp_err= TRUE;
      i++;
    }

    if (cmp_err)
      printf("compare error at %d\n", i);
    else if (rcw ≠ FILE_SIZE)
      printf("Only %d bytes written\n", rcw);
    else if (rcr ≠ FILE_SIZE)
      printf("Only %d bytes read\n", rcr);
    else
      printf("Sent - received comparison ok\n");



    return 0;
} /* end of main() */
```

C Function Descriptions

This section describes in detail how each function is implemented. It is not necessary to read this section to use the library; the manual pages in appendix A should be sufficient for that purpose.

C.1 PFILE *pfopen(char *filename, char *mode, int last, int *blk_size)

There are two different ways **pfopen** performs, depending on the mode. If a new file is to be created (the "w" option), then all the stripe files are created with a **fopen** command, truncating any existing files to zero length. The parameter *last* determines how many stripe files are needed. Then the descriptor file is created in the same manner.

If the parallel file already exists, and mode "r" is specified, several consistency checks are made.

1. The descriptor file is read in and the magic number is verified to make sure it actually is a descriptor file.

2. We already know the start disk of the parallel file from the *filename*. *last* is read in from the descriptor file, ignoring any user supplied value, and then checked, to make sure it falls within the allowed range.

3. Now the length of all existing stripe files are compared to the numbers stored in the descriptor file. This check is necessary to make sure the information in the descriptor file has been updated after the last parallel file operation. This can be done with the **pfflush** or **pfclose** functions.

   If the numbers match, we can be reasonably sure the *current* field in the descriptor file is correct. This value tells us where (what disk) to append data to the parallel file.

The descriptor file (as well as the stripe files), is left open until **pfclose** is called. During a **pfwrite, pfread**, and a **pfflush** the descriptor is updated and written to the descriptor file.

pfopen also mallocs memory for the **PFILE** structure. A pointer to this structure is returned when all tests have been passed successfully. Otherwise, an error number is stored in *pfs_error*, and NULL is returned.

The **PFILE** structure contains the **FILE** pointers to each open stripe file as well as the descriptor file. The structure also contains information about the first and last disk used, the block size, and the current file pointer position for this parallel file.

C.2 int pfclose(PFILE *pfile)

The descriptor file is updated and then closed. All the stripe files are also closed, and the memory holding **PFILE**, is freed.

C.3 int pfflush(PFILE *pfile)

A **fflush** instruction is issued to all stripe files, the descriptor file is updated and also **fflush**-ed.

C.4 int pfwrite(void *ptr, int size, int nobj, PFILE *pfile)

A user block is written to the various stripe files as outlined in section **??**. If the user buffer is not word aligned, one, two, or three bytes are sent ahead before the bulk transfer begins. Since the block size is usually a multiple of a word's length, this alignment has to be made at the start of every block. Users are, therefore, discouraged to read and write from unaligned buffers.

Even when aligned buffers are used, the above checks and procedures may have to be applied. If the previous **pfwrite** was for an uneven number of bytes, the filling of the last stripe in the next **pfwrite** operation will leave the internal buffer pointer at an uneven address. Users should therefore make sure that *nobj * size* is a number evenly divisible by four. This assures best performance.

C.5 int pfread(void *ptr, int size, int nobj, PFILE *pfile)

pfread performs very similarly to **pfwrite**, except data flows into the opposite direction.

C.6 int pfremove(char *filename, int last)

First, the descriptor file is opened and read. From it, the necessary information to find all the stripe files, is extracted. If that works, the stripe files and the descriptor file are deleted. *last* is ignored in that case.

If the descriptor file is, for any reason, unreadable, the parameter *last* is used to find the last of the stripe files. Since the stripe files and the descriptor files are ordinary files, the danger exists, that one of them is inadvertedly deleted. The *last* parameter allows us to delete all parts of the parallel file, even if the descriptor file has been corrupted.

C.7 int pfrename(char *oldname, char *newname)

Renames all parts of a parallel file to a new name. If one of the **rename** functions fails, an attempt is made to go back to the state before the call to **pfrename**. If,

in the meantime, another process has created a file whose name now interferes with the "undo" operation, **pfrename** has to abort.

In a later version this problem could be avoided by first attempting to create empty files of the given new name. Then all files are "copied" to the new ones, and if everything so far has worked, the old names will be removed.

If *newname* contains the name of a directory, that directory must exist on all disks the parallel file uses. Also, the controller number in *oldfile* and *newfile* must be identical; i.e. a relocation from one controller to another is not supported. A lateral move, say from disk 0 to disk 2 is allowed however, as long as all necessary directories exist on all disks affected.

C.8 int pferror(PFILE *pfile)

returns TRUE if the error flag in the **PFILE** structure is set. This is a macro in *clparfs.h*.

C.9 void pfclrerr(PFILE *pfile)

This macro clears the error bit in the **PFILE** structure.

C.10 int pfeof(PFILE *pfile)

The last macro in this group checks the EOF bit in the flag of the **PFILE** structure.

C.11 void pfperror(char *str)

A string describing the error stored in the global variable *pfs_error*, is displayed on stderr. If *str* is not NULL, the string it points to is printed first, followed by a colon, followed by the error message.

Should *pfs_error* be zero, nothing at all is displayed. **pfperror** uses **pfstrerror**.

C.12 char *pfstrerror(int n)

This function returns a pointer to a message appropriate to the error number in n.